JAVM: Internet-based Parallel Computing Using Java

L. F. Lau, A. L. Ananda, G. Tan, W. F. Wong
School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543
laulf@comp.nus.edu.sg

The JAVM (Java Astra Virtual Machine) project is about harnessing the immense computational resource available in the Internet for parallel processing. In this paper, the suitability of Java for Internet-based parallel computing is explored. Next, existing implementations of systems that make use of Java for network parallel computing are presented and categorized. A critique of these implementations follows. Basing on the critique, the requirements and goals of an effective parallel computing system in the Internet environment are singled out. These serve as the blueprint for the development of the JAVM system. Its infrastructure and features, namely *ease of use*, *heterogeneity*, *portability*, *security*, *fault tolerance*, *load balancing*, *scalability* and *accountability*, are discussed. Lastly, experimental results based on the running of several parallel applications in the JAVM environment are presented. Basing on the results, the kind of parallel applications that would be well suited for running in JAVM are identified.

1. Introduction

The exponential growth of Internet in recent years has linked tens of millions of computers together. The combined computational power of even a small fraction of this pool of machines, which are idle most of the time, is many times more than what a central parallel supercomputer can offer. In fact, there have been many successful attempts to solve complicated mathematics and scientific problems by tapping this gigantic computational resource. A more notable recent attempt would of course be the cracking of the 56-bit DES using "tens of thousands" of Internet-linked computers in 39 days [5]. Another commendable effort is the SETI project [31], which has made use of 1.3 million computers worldwide for intensive data processing in its search for extraterrestrial intelligence. Since it inception in May 1999, it has already clocked up 100,000 years of computer time.

However, harnessing the resources of Internet-based computers for parallel computing, is by no means an easy task. It introduces new difficulties and problems that have never been addressed by parallel computing in LAN (local area network) environment.

1.1 Project Objectives and Goals

Java Astra Virtual Machine (JAVM), an extension of the AVM project [19], is developed to provide Internet-based parallel computing with the objective of addressing the difficulties that such a computing environment might impose. In particular, the project aims to:

- Create an infrastructure to support parallel computations in an Internet-based environment. The design and implementation of this infrastructure are based on the following goals:
  - Ease of Use
  - Heterogeneity and Portability
  - Security
  - Fault Tolerance
  - Network Load Balancing
  - Scalability
  - Accountability

- Provide programming interface that could enable programmers to develop parallel applications with ease.

2. Java

Java, a programming language designed from ground up with networking in mind, is set to become one of the most important and widely used languages in the Internet era. Its popularity has surged with the phenomenal growth of Internet. With its cross-platform, secure, object-oriented, and network-centric features, it is ideally suited to address the issues of parallel computing in Internet-based environment. These features offered by Java are lacking in traditional programming languages like C, C++ and Fortran [24].

2.1 Ease of use as network programming language

As described in [9], it is "far easier to write network programs in Java than in almost other language". As a network programming language, besides supporting TCP/IP socket programming, Java also incorporates useful features such as object serialization and RMI (Remote Method Invocation). Object serialization allows objects to be written to, and read from streams (such as file streams or socket data streams) as easily as numbers and characters. RMI uses object serialization for storing and communicating with objects. Together, they allow Java programs to call certain methods on a remote server, with minimal effort.

With these features, in the development of network applications, the complexity of the underlying network communications is hidden from the developer. Most of network-related work is done automatically and the developer can just concentrate his effort on developing the functionality of the applications.

2.2 Platform Independence and Code Mobility

Java is a platform independent language. As such, in the development of applications, it is able to allow programmers to "write once, run anywhere". Java programs can be developed in any platform and complied into a single set of executable binaries (Java bytecodes). These binaries, which can exist in the form of applets, are download-able from web servers to any Java-enabled browser in client machines, of any platform,

and executed there. This ability to transfer executable binaries to wherever they need to be executed is known as code mobility.

In traditional message passing systems, like PVM (Parallel Virtual Machine) and MPI (Message Passing Interface), the program to be executed has to be resident in each machine and the user will need to have an account in each machine participating in the computation. For instance, to perform a parallel computation using PVM, the following steps must be performed:

1) Install PVM daemon on all participating machines.
2) Compile the application binaries for each target architecture.
3) Distribute binaries to all machines, either by explicit copying or by using shared file system.
4) Provide remote shell access to user so as to allow remote execution of the PVM daemon and application binaries.

For an Internet-based computing environment formed by machines of heterogeneous platforms, the above procedure is tedious and unacceptable. With Java's platform independence and code mobility, none of the steps in the procedure needs to be performed.

2.3 Security

Java is a language designed with security and safety in mind. At the language level, it has several features to protect the integrity and security of the system. As listed in [25], those features are:

- Access controls on variables and methods in objects.
- Declaration of final classes or methods
- Type safety checking
- Elimination of pointer arithmetic

The security model adopted by Java is referred as the *sandbox* model [17, 25]. This model combines *bytecode verifier*, *classloader* and *security manager* to create a secure environment whereby untrusted Java codes can be executed with the assurance that the security of the environment is not compromised. An illustration of this security model is shown in Figure 2.1.

The *bytecode verifier* ensures that only legitimate Java code is executed. Together with the Java virtual machine, they guarantee language safety at run-time. The *classloader* defines a local name space such that untrusted java codes cannot interfere with the running of other Java programs. The *security manager* enforces security policies for executable content. Based on certain predefined policies, it controls access to crucial system resources and the actions that a particular Java class can perform.

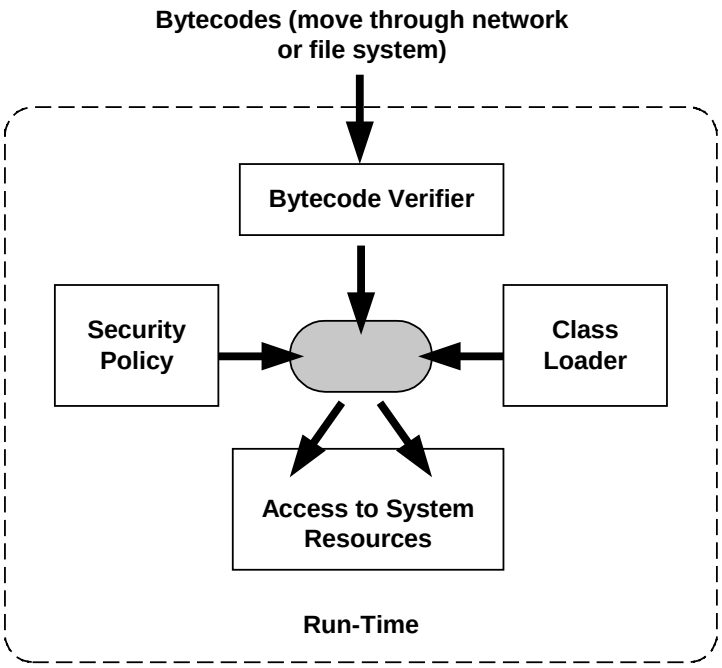

Figure 2.1. Java Security Model

Due to Java's security features, Java applets loaded from network cannot perform (as listed in [9]) the following:

- Accessing arbitrary addresses in memory.
- Accessing local file system in any way.
- Launching of other programs on the client machine.
- Loading of libraries or defining native method calls.
- Revealing of information of the user or the user's machine, such as user name or home directory.
- Defining of any system properties.
- Accepting of connections from a client machine; an applet cannot create a server on the client.

In the context of Internet-based computing, security is important. Allowing a piece of untrusted codes to be executed on a machine without protection is almost equivalent to giving a stranger free access to the machine. Volunteers participating in parallel computations must be given the assurance that their machines are protected against malicious attacks (e.g. virus attack) and theft of confidential information. Java's security features address this concern.

2.4 Concerns

However, Java's security model is double-edged. Security restrictions of the applets may have blocked out malicious attacks on client machines but it has also set limitations on the kind of network communications that can be performed. As an applet can only open network connections to the server from which the applet was

downloaded, only master-slave style of parallelism can be supported. In fact, as pointed out in [9], if more access to network is required, the program should be developed as a standalone Java application, instead of an applet.

Java's performance is also an important concern. Being a bytecode-interpreted language, Java is a few times slower than other fully compiled languages, like C. However, with code tuning and optimizing, and adoption of techniques like JIT (Just-In-Time) code generation, performance of Java is bound to improve [10]. The recent introduction of Java HotSpot Performance Engine by Sun Microsystems is another booster to Java's performance. It was claimed that HotSpot could improve certain application performance by 100 percent [16].

3. Parallel Computing Systems Using Java

With the powerful features of Java, it is little wonder that a significant number of systems have been developed to support network parallel programming using Java. Basically, these systems can be divided into two categories:

- Based on Java applets that execute within the context of a web browser.
- Based on standalone Java applications.

3.1 Java Applets Implementations

In these systems, applications are broken down into smaller tasks and downloaded, in the form of applets, by client machines volunteering their computational resources. This is simply done by directing the web-browsers at client machines to the URLs of the web servers where the applications are run. After execution at the client machine, the computed results are returned to the server machines where the applets were downloaded. Examples of such systems are Javelin [2], Bayanihan [3,15], Charlotte [1] and DAMPP (Distributed Applet-based Massively Parallel Processing) [14].

Critique

The use of applets has allowed any machine, connected to the Internet and equipped with a Java-enabled web browser, to join any ongoing computation with ease. Due to Java's platform independence, the platform of the participating machine is not an issue at all. Effort required for the setting up of such a parallel computing system, is minimal.

However, due to security restrictions of applets, such systems are only suitable for supporting applications that can be decomposed into non-communicating, coarse-grained and totally independent functional tasks. In short, only *master-slave* style of parallelism can be achieved.

In most of the above systems, the web server is the only contact point for volunteers to download the applets. High congestion, in terms of system and network load, may be experienced at the server and it can easily become a bottleneck for the entire computation. Single point of failure and inscalability of such systems are problems not to be dismissed.

3.2 Standalone Java Applications

Systems, supporting network parallel computing as standalone Java applications, can be sub-categorized into two groups. One that is based on Java Native Methods mechanism and the other that is not.

Native methods refer to methods, called by Java programs, but are written in languages, other than Java. An example of system that uses Java Native Methods mechanism is jPVM [12] (previously known as JavaPVM). By providing native method wrappers around the existing standard PVM routines, jPVM allows Java applications and existing C, C++ and Fortran applications to communicate with one another using the PVM API. Using jPVM, programmers can link existing PVM applications in Java graphical user interface front ends. Another system that uses Java Native Methods mechanism is IceT [4]. However, unlike jPVM, it allows resources on machines to be made available to users who do not have log-on privileges.

Other systems are implemented entirely in Java and make use of the socket interface in the standard API for communication among parallel units. One example is JPVM [11]. It is not interoperable with PVM but provides Java implementation of routines found in the PVM library. Another example is MPIJ [18], a pure Java implementation of Message Passing Interface standard.

Critique

Systems, implemented as standalone Java applications, do not have the same restrictions in network communications as in the case of systems using Java applets. Peer-to-peer communications among participating machines are possible. Those that make use of native methods mechanism also have a clear advantage of software reuse and performance.

However, the use of native methods assumes that the underlying platform already has support for the native methods. Consequently, systems based on native methods mechanisms will face portability problem. For instance, jPVM can only be used on platforms where standard PVM is available.

Implementations of the above systems, with the exception of IceT, only allow users to utilize resources of those machines where they have access privileges. Usually, certain assumptions of the underlying file systems are also made. Consequently, the setting up of such systems for large-scale parallel computations, based on computing resources from volunteers in the Internet, would be an uphill task.

3.3 Weaknesses

Based on the above discussion, the following is a summary of the main weaknesses of existing parallel computing systems using Java:

- Scalability problem.
- Portability problem.
- Single point of failure problem.
- Lack of security mechanisms.

- Lack of fault tolerance mechanisms.

The designs of most existing parallel systems are catered for computing in a small network setup, such as LAN (Local Area Network). The weaknesses identified will handicap their deployment as a parallel computing tool for large scale computations in Internet-based environment.

4. Project JAVM: Design and Goals

JAVM (Java Astra Virtual Machine) aims to create an Internet-based parallel computing environment using the Java programming language. It is a 100 percent pure Java implementation [26] and not based on native methods mechanism. Based on our classifications of existing implementations of parallel computing systems using Java in Section 3, JAVM fits into the second category of standalone Java application.

As discussed, existing implementations, be it applet-based or standalone application-based, are still lacking in certain features that make their deployments for large-scale Internet-based parallel computing questionable. Certain systems, like jPVM and MPIJ, are not designed for Internet-based computing in the first place. JAVM is for Internet-based parallel computing and aims to address these deficiencies. To achieve this, the following factors have to be addressed:

- Ease of Use
- Heterogeneity and Portability
- Security
- Fault Tolerance
- Network Load Balancing
- Scalability
- Accountability

4.1 Ease of Use

This involves two groups of users – the programmers who develop the parallel applications and the volunteers who contribute their machines for the computations.

From a programmer's perspective, the programming interface must be flexible and simple. Developing parallel applications should be done with as much ease as sequential applications. In JAVM, we aim to achieve this by providing a set of features-rich networking APIs that will shield the programmer from the complexity of the underlying network communications. The collection of heterogeneous machines in the network would appear to him as a single high-performance parallel machine. This will be discussed further in the next section.

From a volunteer's perspective, contributing his machines to participate in computations should be simple and hassle-free. No technical knowledge of the program to be run should be required on his part. As mentioned earlier, in some Java-based computing systems, to volunteer simply means visiting a web server site using a Java-capable browser. However, due to communication restrictions of applets, such an approach is not adopted in JAVM. Instead, users are required to run a program at their machines to join the pool of volunteers. Such an approach is no more difficult

than visiting a website as no other effort is required on the part of the users. Figure 4.1 shows the graphical interface used by volunteers for contributing the resources of their machines in JAVM.

With Java's code mobility, JAVM also supports dynamic loading of applications from the network to be executed at the volunteers' machines. The loading and execution of applications are transparent to the volunteers. It is very much like the dynamic loading and execution of applets, but without their limitations.

4.2 Heterogeneity and Portability

The assumption that machines participating in a computation in a COW (Cluster of Workstations) setup are of homogeneous platform cannot be applied to the Internet context. The Internet is a collection of machines of heterogeneous platforms and architectures. Thus, it will be an uphill task for the programmer to write codes for different systems and distribute the compiled binaries to the systems.

Java is architecture neutral. By virtue of this nature, JAVM, being a 100% pure Java implementation, is portable and will operate faultlessly on any platform as long as Java JDK1.2 is supported by the platform. As of now, JAVM has been tested to work successfully on the following platforms:

- Win32 Version. For Windows 95, Windows 98 and Windows NT 4.0 on Intel hardware.
- Solaris/SPARC Version. Only Solaris versions 2.5.1, 2.6 and 7 (also known as 2.7) are supported.
- Solaris/Intel Version. Only Solaris versions 2.5.1, 2.6 and 7 (also known as 2.7) are supported.

4.3 Security

This involves two aspects. Firstly, from a volunteer's perspective, by contributing his machine in a computation, he must have the assurance that it will be protected against malicious attacks (e.g. virus attack). In a COW setup, this is easily by maintaining users access rights and accounts, so that the network is available only to trusted users. Using such an approach, security of volunteer's machine is enforced to a certain extent, but the setting up of such a computing environment would be tedious and introduces problems, such as scalability. In JAVM, no assumptions of access privileges of machines participating in computations and their underlying file system are made. Instead, as already mentioned in Section 2, protection of volunteers' machines can be enforced with Java's tight security model.

The second aspect involves computational security. Besides protecting volunteers, the environment must also ensure that the computation itself is secure from malicious attacks from volunteers. Preventing volunteers from *spying* on computations, which may be confidential, is also another concern. In JAVM, enforcing secure computations is a prime concern. To achieve this, we implement cryptographic and authentication mechanisms for network communications among machines in the system. This topic will be covered in details in next section.

4.4 Fault Tolerance

An Internet-based computing environment is governed by the following characteristics:

1) Uncertain availability of machines. In the Internet, machines join and leave a computation at the whim of volunteers. There is no direct control as to when a volunteer should join a computation and how long he should remain committed to it.

2) Unpredictable network delay. The load of the Internet traffic and faulty network links are the major contributing factors.

3) Unreliability of volunteers. Based on what we have discussed on security, the reliability of the computed results from volunteers' machines is questionable.

The JAVM system is designed to recover from unintentional faults caused by the first two characteristics. It polls machines in the system at regular interval to check whether they are still "alive". In the event that a volunteer leaves the system abruptly, JAVM is able to off-load the failed computation to another machine such that the overall computed results would not be affected. This implementation will be elaborated in the next section.

Intentional faults caused by the third characteristic are harder to tackle. Known techniques, such as *replication* and *spot-checking* [15], to detect and recover from these faults are inefficient and unreliable. In JAVM, at present, no solution is offered to counter this problem yet. Instead, through tight security enforcement, occurrences of such faults caused by rogue volunteers are minimized, if not avoided.

4.5 Network Load Balancing

Load balancing usually refers to the distribution of workload evenly among participating machines. However, in the Internet context, this is insufficient and the traffic load of network links should also be taken into account. A system with support for network load balancing should be able to detect congested network links and avoid assigning computations to machines reachable by those links, whenever possible. To achieve this, in JAVM, all the volunteers in the system are polled at regular intervals to measure the network transfer delays. We called this technique *bandwidth probing*. This technique will be discussed again in details in the next section.

4.6 Scalability

To tap the computational resources of tens of thousands of machine in the Internet for parallel computing effectively, the scalability of the system is an important issue. The communication architecture (or topology) supported by the system, is a deciding factor in its scalability. The merits and demerits of the different forms of

communication architecture (e.g. centralized and decentralized) are discussed in [6]. In JAVM, for scalability reason, the architecture adopted is a hybrid between the centralized and decentralized one. The main benefits of this architecture are its abilities to reduce message traffic exchanges (as compared to a decentralized design) and improve fault-tolerance (as compared to a centralized design). An illustration of this communication architecture is given in Figure 4.2 and a detailed discussion of this topic, in Section 4.2.

4.7 Accountability

In JAVM, an accounting system is put in place. Volunteers contributing their resources will be duly "rewarded" and clients consuming the resources, "billed" accordingly. The architecture of the JAVM system permits the implementation of this feature to be done quite easily. It involves the maintaining of databases at centralized locations to capture the contribution and usage of computing resources by entities in the JAVM system.

4.8 Network Communication Architecture

The design of the network communication architecture of a computing system will directly, as well as indirectly, affect the implementation of the design goals listed. As shown in Figure 4.2, JAVM adopts hierarchical network communication architecture consisting of four entities: *director, coordinator*, *volunteer* and *client*. To prevent the diagram from getting too complicated, the direct communication links between volunteer and director, and that of client and director, are not shown.

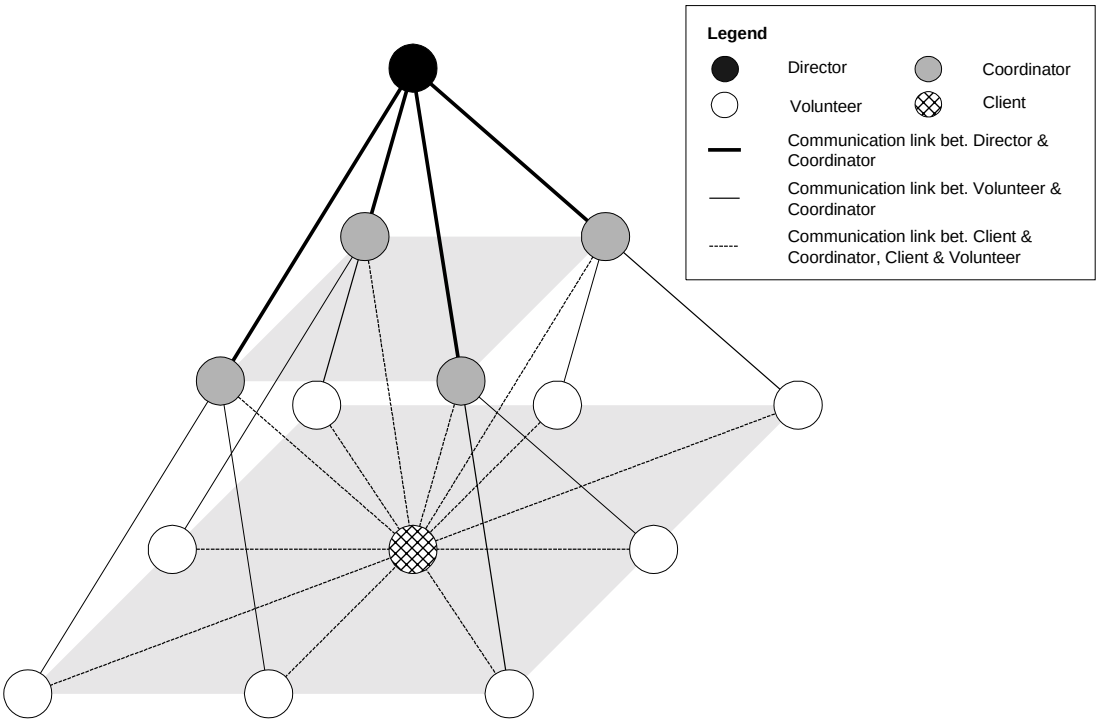

Figure 4.2. Network Communication Architecture of JAVM

The main function of the director is to provide lookup service for coordinators information in the JAVM system. As the name implies, coordinators serve as the middlemen between volunteers and clients during task distribution. Volunteers are machines that offer their resources for parallel computations. Clients are machines that tap these resources.

4.9 Scenario Walkthrough

A typical scenario, of how the entities might interact with each other, is shown in Figure 4.3.

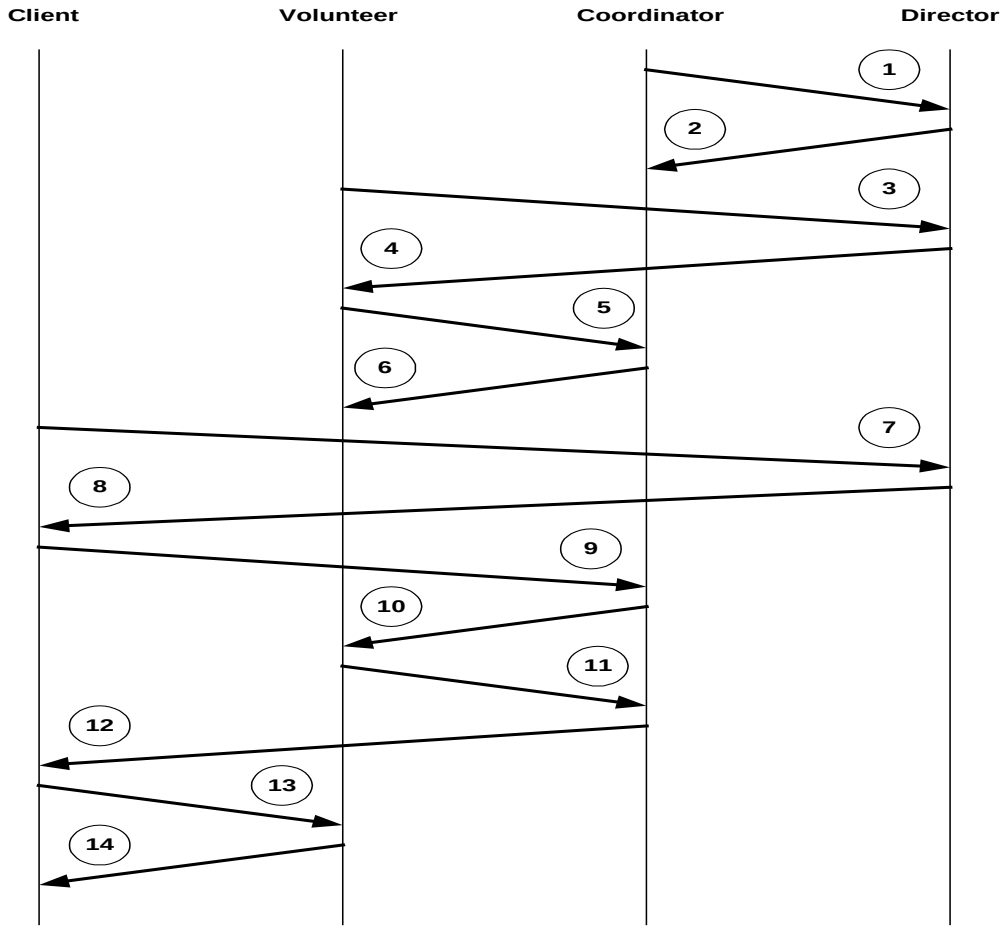

Figure 4.3. Interaction among JAVM entities

The sequence of the interactions is as follows:

(1) When a coordinator first started up, it will register with a specified director in the JAVM system. In its registration request to the director, it will supply a password for verification purpose.

(2) On receipt of the registration request, the director will check the information supplied by the coordinator against its database of legitimate coordinators. If the coordinator passes the check, it will be added to the director's list of active coordinators.

(3) When a volunteer first started up, it will contact the director for information of the active coordinators in the JAVM system.

(4) The director returns the list of active coordinators that have registered with it.

(5) From the list, the volunteer selects one coordinator, by random, for registration. In its request to the coordinator, the hardware information and lease time for volunteering is supplied.

(6) On successful registration, the coordinator will add the information supplied by the volunteer to its database of registered volunteers. A random number (called the volunteer-id) will be generated, and the volunteer, informed. This id will be used in subsequent communications between the volunteer and the coordinator for identity checking purpose.

(7) When a client first started up, it will contact the director for information of the active coordinators in the JAVM system.

(8) The director returns the list of active coordinators that have registered with it.

(9) The client will contact the first coordinator, in the returned list from director, for volunteers. In its request, it will supply a password for verification purpose and also the number of volunteers needed. If insufficient volunteers are returned, it will contact other coordinators in the network until all the volunteers required are found.

(10) On receipt of the request for volunteers, the coordinator will check the information supplied by the client against its database of legitimate clients. If the client passes the check, the coordinator will select a list of potential volunteers (based on the number of volunteers requested) for allocation to the client. For each of these volunteers, a random number (called the client-id) is generated. The coordinator will then inform every volunteer of its allocation and the client-id to be used during interactions with the client.

(11) The volunteer will acknowledge the receipt of the allocation notice by sending a confirmation reply to the coordinator.

(12) The coordinator will send the list of allocated volunteers and their corresponding client-ids to the client.

(13) With information received from the coordinator, the client will send task placement requests to the volunteers. Information, included in the requests, is the client-ids, arguments and bytecodes of the application to be run.

(14) The volunteer will check whether the client-id supplied by the client matches that given by the coordinator earlier. If they match, the volunteer will “class-load” the bytecodes supplied for task execution. Result of the execution is then sent back to the client.

The roles, relationships and interactions of the entities will be further elaborated in subsequent sections.

4. 10 Constraints and Limitations

In general, the computing models for parallel applications can be classified into two categories: *crowd computing* and *tree computing*. The crowd computing model involves a collection of closely related processes, typically executing the same code, perform computations on different portions of the workload and usually involving the periodic exchange of intermediate results. Master-slave parallelism is an example of such a computing model. The tree-computing model, as implied by its name, involves the spawning of processes in a tree-like manner, thereby establishing a parent-child relationship. Such a model, though less commonly used, is a natural fit for applications that use "branch-and-bound" algorithms and recursive "divide-and-conquer" algorithms.

For now, the design of the JAVM system can only achieve master-slave style of parallelism. This implies that volunteers participating in the execution a parallel application would not be able to communicate with each other during computation. Consequently, JAVM can only applications that can be broken down into non-communicating and independent functional tasks. However, as JAVM is a non-applet based implementation, it can be enhanced further to support peer-to-peer communications among volunteers and parallel applications based on tree computing model.

Communication between two entities in JAVM system involves encryption of data at the sending party and decryption by the receiving party. In addition to this, overhead caused by network transfer of data between the parties will also have a negative impact on the overall performance of the application. To offset the impact caused by the overheads, the tasks sent to volunteers during task placement should be of large-grained size. This point is substantiated by experimental results in Section 7.

As mentioned earlier, intentional faults caused by rogue volunteers cannot be tackled by the JAVM system, at this moment. Enhancement to the JAVM system in this area should be made.

5. Project JAVM: Design

This section describes the implementation of the JAVM system. Details of algorithms used in fulfilling the design goals identified in section 4 are given.

5.1 Scheduling Algorithm

As mentioned, one of the goals of JAVM is to support network load balancing. To achieve this, JAVM implements a scheduler such that volunteers with the highest capability are selected for task execution. The description of the scheduling algorithm will be based on the *transfer policy*, *information policy* and *placement policy* adopted.

5.1.1 Transfer Policy

This policy decides when and how an attempt should be made to transfer an arriving task to another host.

The client will have *n* number of tasks for distribution to volunteers in the network for processing. It will contact a coordinator in the JAVM system for *n* number of volunteers for the processing. The coordinator will search its database of registered volunteers and select *n* (if possible) most capable volunteers for allocation to the client. However, under circumstances when the coordinator is unable to meet (either fully or partially) the client's demand, the client will contact other coordinators in the network for the remaining number of volunteers required. The process repeats itself until all *n* volunteers are found.

5.1.2 Information Policy

This policy is concerned with deciding what kind of information is to be collected, when, where and how it should be collected for scheduling purpose.

During registration with coordinator, a volunteer will supply its hardware information (CPU speed, memory capacity and disk capacity) and the lease time for contributing its resources. The coordinator stores this information in a *VolunteerNode*. The coordinator also maintains a linked list of *VolunteerNode*s (known as the *VolunteerList*) that contains information of all other volunteers that have registered with the coordinator. At regular intervals, the coordinator will do *bandwidth probing* on every volunteer by prompting them to send certain length of byte stream over. The network transfer time (called *netDelay*) of each volunteer is measured from the moment that the volunteer is prompted till the moment the byte stream is received by the coordinator. Failure of a volunteer to respond to the coordinator's prompting will signify that it has been disconnected from the network. Consequently, information of the failed volunteer will be removed from the linked list.

Information stored in a *VolunteerNode* is as follows:

1) InetAddress of volunteer
2) Network transfer of byte stream, *netDelay*
3) CPU speed of volunteer
4) Memory capacity of volunteer
5) Disk capacity of volunteer
6) Lease time for volunteering resources. Expiration of this lease indicates the exit of the volunteer from the JAVM system.
7) InetAddress of client which the volunteer is assigned to, *assignedClient*
8) Time when the assignment is made, *assignedTime*

It is clear that information of all the volunteers in the network is not kept centrally in one coordinator. With a distributed design by use of multiple coordinators, single point of failure is avoided. With different clients consulting different coordinators for volunteers, the likelihood of any coordinator becoming a bottleneck is reduced.

Note also that the coordinators do not exchange information of its volunteers among themselves. This is to reduce communication traffic among the coordinators. The JAVM system is designed for Internet-based computing with large number of participating volunteers. If all the coordinators were to update each other of every change of network status of each of their volunteers, the frequency of communication exchanges will be very high.

5.1.3 Placement Policy

This policy decides the host to which a task should be transferred for execution.

$$\mathbf{CP = W_1*CPU_N + W_2*MEM_N + W_3*DISK_N + W_4*NET_N}$$

where CP stands for Capability value of the volunteer

W_1, W_2, W_3, W_4 are weightages of the respective resources. To be decided by coordinator.

NB: W1 + W2 + W3 + W4 = 1

CPU_N, MEM_N, $DISK_N$, NET_N are the normalized values of the CPU speed, memory capacity, disk capacity and network load of the volunteer respectively.

NB: Each value is derived from the normalization of the resource value against the best possible value in its category.

Figure 5.1. Capability Value Derivation for Volunteer

Upon receipt of requests from clients for volunteers, the coordinator will select the best volunteers in its *VolunteerList* for tasks placement. Only volunteers with the top *capability* values will be selected. The derivation of capability value for each volunteer is based on *netDelay* (the network delay information collected from bandwidth probing), and the hardware information of the volunteer (collected during registration). The formula used is as shown in Figure 5.1.

The *VolunteerList* is sorted, based on the *capability* value of each *VolunteerNode*, in an ascending order. This is to facilitate the selection of the best volunteers during task placement. The reasons for choosing network bandwidth as one of the parameters for placement policy are as follows:

1) In the JAVM system, the participating hosts may be geographically distributed in different parts of the world. Consequently, the network communication delay among the hosts will be quite significant, as compared to a LAN (Local Area Network) setup. In view of this, if system load were to be used as a factor for volunteer selection, the load information from the volunteers that arrives at the

coordinator may be obsolete and may not truly reflects the load state of the volunteers at that point in time.

2) In the JAVM system, bytecodes of the applications from the client will be transferred to the selected volunteers for execution. This may involve a huge amount of data transfer. If the links to the volunteers are congested or of low bandwidth, the transfer may take up a significant amount of time and hence, defeat the purpose of distributed processing.

3) Although the *netDelay* is a measure of the link status between a volunteer and the coordinator, it can also be viewed as an indicator of the load of the volunteer. A lower loaded host will respond faster than a heavier loaded one during bandwidth probing. Therefore, the lower *netDelay* value of the volunteer recorded at the coordinator will increase its chance of selection for task placement, over its heavier loaded counterparts.

The use of the formula for deciding the best volunteers for task placement allows the deployment of JAVM in different environments. In the Internet environment, which JAVM is designed for, more emphasis should be put on the weightage for the *netDelay* parameter. However, for deployment in a LAN environment, whereby the network is more stable but participating machines are of varying hardware capabilities, the weightages for system resources (*CPU*, *MEM* and *DISK*) should be increased and that for *netDelay*, decreased accordingly.

5.2 Class Loading

The heterogeneous nature of the Internet does not allow JAVM to make any assumption of the underlying file systems and access privileges to systems. As such, to utilize the resources of a volunteer machine, classes (in the form of bytecodes) of the task to be executed will have to be dynamically transferred from the network to the volunteer during run-time. This mobility of codes is achievable in Java by a mechanism called *class loading*.

In JAVM, we have implemented a class loader, called *JAVMClassLoader*, to take care of the loading of task bytecodes, from the client across the network, and dynamically linking them with the run-time system of the volunteer. The *JAVMClassLoader* starts by being a subclass of **java.lang.ClassLoader**. An overview of this implementation is as follows:

- Using the default system class loader, check if the class to be loaded is a system class.
- If it is not, attempt to fetch the class from the classloader's class repository.
- If failed, get the bytecodes of the class from the *byteCodesTable*, which is sent by the client across the network.
- Perform bytecodes verification and convert the bytecodes into a class object.

With the help of the *JAVMClassLoader*, the volunteer will load all the necessary classes needed for the task execution. Upon completion, the final result obtained is then returned back to the client.

5.3 Security

The Internet is an uncontrolled network and has long been considered an insecure environment. Stories of security violations such as machines being hacked and network communications path being tapped, which resulted in the loss or theft of important data, are rampant.

JAVM aims to provide an Internet-based computing environment that ensures security. On top of the existing protection provided by the Java language itself, JAVM offers encrypted communication exchanges among coordinators, volunteers and clients. Every host in JAVM possesses a common *secretkey* that is used for the encryption and decryption of messages. Hence, even when the communication path is tapped, the information that could be viewed by the intruder is useless.

JAVM has also implemented authentication by the use of identification numbers and passwords, which the sending hosts must provide to the receiving hosts. It is only after verification that the identification numbers and passwords match with those at the receiving hosts that the incoming connections are accepted. With this feature, computational resources of volunteers are guarded against unauthorized clients.

5.3.1 JCA/JCE and the DSTC Security Provider

Before we delve into the details of the cryptographic mechanisms used in JAVM, let's talk about the security API provided in Java.

The Java Cryptography Architecture (JCA) provides a standard set of APIs for cryptographic operations. As noted in [21], to ensure the highest degree of flexibility for both the developer and the end user, the JCA places great emphasis on algorithm independence and implementation independence. To achieve these two requirements, the design of the Java Cryptography API is based on a system of engines (also known as factory methods) and providers. Details can be found in [21,22,23]. JCA adopts a Service Provider architecture whereby it is easy to install, configure and use third party implementations of cryptographic algorithms.

Because of US export restrictions, Sun split its cryptographic classes into two groups. The first group, **java.security.*** packages, forms part of JDK 1.2 and can be exported without restriction. The second group, the Java Cryptography Extension (JCE), is only for distribution within US and Canada. The JCE is a standard extension library to JCA supporting encryption and key agreement operations.

Due to the US export restrictions, we are not able to use Sun's JCE implementation for the cryptography required in JAVM. Fortunately, Distributed Systems Technology Centre (DSTC) provides a clean room implementation of JCE 1.2 and a service provider (the DSTC Provider) [20]. The implementations from DSTC are adopted for the cryptography needs in JAVM.

5.3.2 Cryptography in JAVM

In JAVM, we make use of DES (Data Encryption Standard), a symmetric cipher, for the encryption of communication messages. DES was first published in 1975 and has withstood intense cryptanalytic scrutiny since then. The major weakness of DES is its use of 56-bit key size, which makes it vulnerable to key search attack.

For secure communication in JAVM, a message at the sending party is first stored as a *serializable* object. Before sending out to the receiver, the object is encrypted by making use of the **javax.crypto.SealedObject** utility class in JCE 1.2. The *SealedObject* constructor wraps around the supplied serializable object and encrypts it with a supplied cipher. At the receiving party, the *getObject*() method is used to retrieve the original, unencrypted object with the same cipher.

In JAVM, the *CipherObjectOutputStream* and *CipherObjectInputStream* classes are developed to hide the implementation details of the encryption and decryption processes. Specifications of the two classes can be found in Appendix I.

5.4 Programming Model and Developer Interface

```
public static void main (String[] args) {
        ….
        //  (1) Create the environment
        JAVM_Env env = new JAVM_Env(className, appnClassNames,
                                        "SecretKey.ser", director, password);
        ….
        //  (2) Generate tasks to TaskPool
        for (int i=0; i<numVolunteers; i++) {
          ….
          tidarray[I] = env.AddToTaskPool((Object) task)
        }
        ….
 //  (3) Distribute tasks to volunteers
 env.SendTasks();
 ….
 //  (4) Collect computed results
 for (int i=0; i<numVolunteers, i++) {
   ….
   result[i] = env.getResult(tidarray[i], waitTime);
 }
 ….
  // (5) Stop all unfinished tasks at volunteers
  env.StopTasks();
     }
```

Figure 5.2. JAVM Programming Model

As mentioned, one of the objectives JAVM is to provide programming interface that can help developers to program parallel applications with as much ease as sequential ones. Figure 5.2 gives an illustration of how simple developing parallel applications in JAVM could be. The APIs called by the developer hides all the implementation details of encryption and decryption of network communications, fault tolerance and

network load balancing. To the developer, the pool of volunteers in the JAVM system will appear as just one virtual supercomputer.

The programming model in Figure 5.2 encompasses the following steps:

(1) Create the environment. In the background, the client contacts the director for the available coordinators in the JAVM system. An empty linked list (called the *TaskPool*) is also generated for the storage of tasks that will be generated.

(2) Generate tasks to *TaskPool*. The number of tasks generated depends on the number of volunteers specified by the developer for execution of the application. The tasks generated, in the form of arguments for the application, are stored in the *TaskPool*. For each task created, the developer will be issued a *task-id*, which is needed in the retrieval of result at stage (4).

(3) Distribute tasks to volunteers. In the background, the client contact the coordinators to get the volunteers needed. It will then transfer the bytecodes of the application and the respective argument to each of the volunteers found by invoking the *TaskPlacer* thread. The client might not be able to get all the volunteers it needs and the existing volunteers allocated might leave the system abruptly without finishing the task allocated. As such, the client will invoke the *VolunteerHunter* thread, which will hunt for new volunteers from the coordinators whenever there is a shortage. It will also invoke the *VolunteerAliveHandler* thread to check, at regular intervals, whether the volunteers allocated with tasks are still alive. Detection of failed volunteers will spin off the whole process of finding new volunteers and transferring of tasks, once allocated to those failed volunteers, to the new volunteers. This recovery mechanism is automatic and transparent to the developers. At this stage, for collecting results back from the volunteers, the client will also invoke the *ResultCollector* thread.

(4) Collect computed results. Results computed by the volunteers will be received by the *ResultCollector* thread and stored. The coordinator will also be informed to release allocations of the volunteers held by the client. To retrieve the computed results, the developer will have to supply the necessary *task-id*s issued earlier at stage (2).

(5) Stop all unfinished tasks at volunteers. This step is optional as it depends on the application's need. If the developer has acquired the necessary results, it can terminate existing execution of tasks at all volunteers by invoking the *StopTasks()* method. For applications that need to collect back all computed results, such invocation will not be necessary.

6. JAVM Performance

This section gives descriptions of the experiment setup for measuring the performance of the JAVM system. The experiments conducted are based on parallel applications developed for the JAVM system. An analysis of the results obtained from the experiments is also presented.

6.1 Experimental Setup

The test bed for conducting the experiments is as illustrated in Figure 6.1. For the sake of consistency in the measurement of the experimental results, a homogeneous collection of 64 Pentium PCs, running WinNT OS, is used to form the volunteers cluster. To show heterogeneity of the JAVM system, machines of different hardware configurations and OS (UltraSPARC, Solaris 2.7 OS) are chosen to form the director, coordinators and client. All the machines are connected to the network of School of Computing via 100Mbps links.

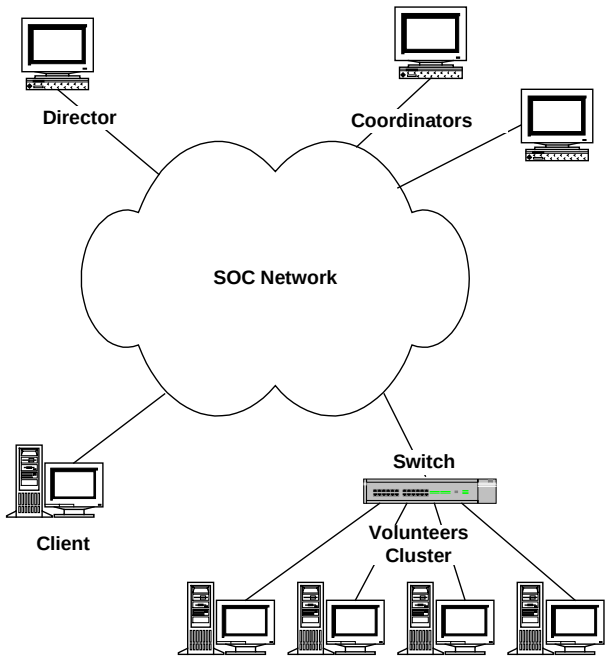

Figure 6.1. Experimental Setup for JAVM Performance Measurement

The details of the hardware configurations of the machines are as shown:

	Processor	**OS**	**RAM**
Volunteer	2 x Pentium III 450 MHz	WinNT 4.0	256 MB
Director	4 x UltraSPARC-II 300 MHz	Solaris 2.7	4 GB
Coordinator	2 x UltraSPARC-II 300 MHz	Solaris 2.7	2 GB
Client	1 x UltraSPARC-II 300 MHz	Solaris 2.7	1.2 GB

Table 6.1. Machine Configurations

For the sake of consistency, the experiments are conducted when the load of all the machines and the network are low.

6.2 RC5 Cracking

RC5 [27] is fast block cipher that uses a parameterized algorithm with a variable block size, a variable key size and a variable number of rounds. Typical choices for the block size will be 32, 64 or 128 bits. The number of rounds can range from 0 to 255 and the key can range from 0 to 2048 bits in size.

The encryption routine of RC5 consists of three primitive operations: integer addition, bitwise exclusive-or, and variable rotation. The heavy use of data-dependent rotations and the mixture of different operations provide the security of RC5.

6.2.1 Experimental Results

For this, a sequential RC5 cracking program [28], for uncovering the 32-bits secret key used to encrypt a message, is ported over to JAVM.

The sequential version adopts a "brute force" search on all possible keys until it finds the one that decrypts the message. Time ($\mathbf{T_{seq}}$) needed to run the program on a machine (with same configuration as the volunteers) is measured. This value is **8.24×10^6 milliseconds**.

The JAVM-ised version breaks up the key space to search into smaller ones (of equal size) and distribute them to the volunteers so that the search can be performed in parallel. For this experiment, the program is run with different number of volunteers in the network. Time ($\mathbf{T_{dis}}$) is measured from the moment the client creates the JAVM environment to the moment when the secret key is found by one of the volunteers and the result returned to the client. The speedup ($\mathbf{T_{seq}/T_{dis}}$) for each of above experiments is then computed.

No. of Volunteers (n)	**Time Taken (T_{dis}) ($\times 10^6$ ms)**	**Speedup**
2	4.108	2.005
4	2.053	4.013
8	1.028	8.011
16	0.529	15.552
32	0.283	29.018
64	0.154	53.426

Table 6.2. RC5 Cracking Experimental Results

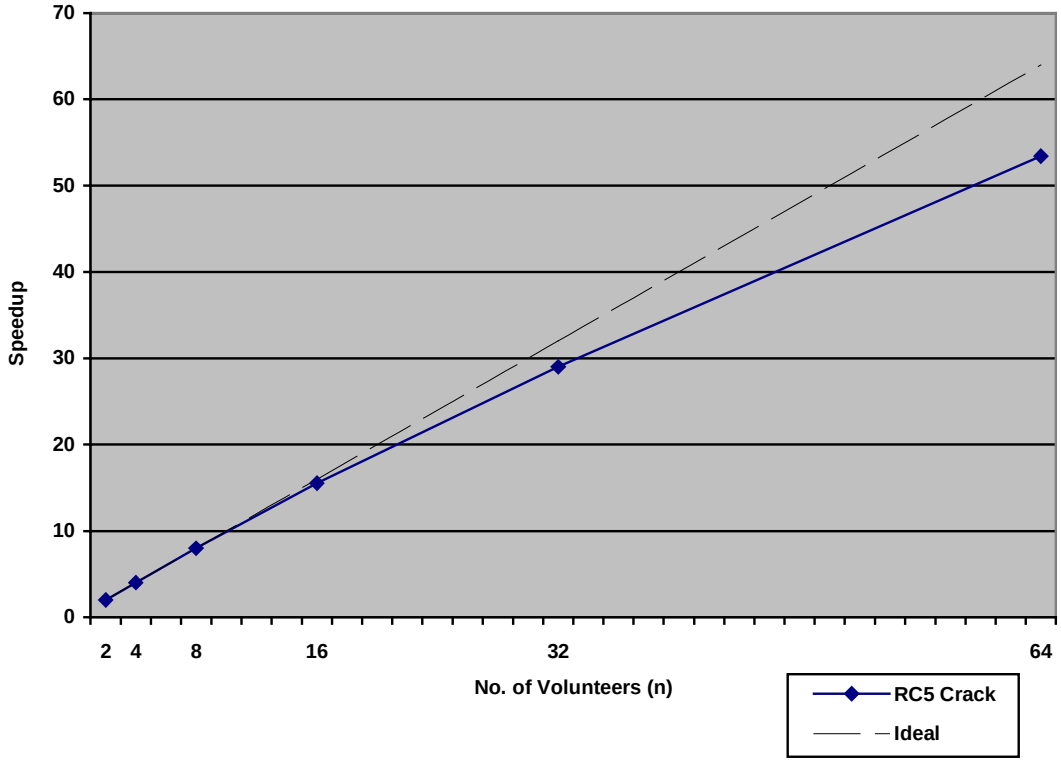

Figure 6.2. RC5 Cracking Experimental Results

6.2.2 Performance

The speedup curve (in Figure 6.2) obtained for JAVM-ised parallel cracking is very close to the ideal situation. Characteristics of this application are as follows:

- Highly computational intensive.
- Low bandwidth requirement. The bulk of the network communications among the entities in the system is only during task distribution and result collection stage. In these stages, the amount of data exchanged is usually not high.

6.3 Ray Tracing

Ray tracing is a method that creates photo-realistic synthetic images from a mathematical description of the scene to be generated. This computer graphics technology simulates light rays in a 3D environment. The algorithm attempts to calculate the exact coloring of each ray-object intersection by tracing light rays as they bounce around the scene, reflecting and refracting until they end up in the lens of the imaginary camera. The computations involved can involve millions of floating-point operations and consume large amount of processor time even with scenes of modest complexity. Fortunately, each ray in the algorithm is independent of each

other, making ray tracing an ideal candidate for parallel processing. As such, time taken for image processing can be drastically shortened.

6.3.1 Experimental Results

For our experiments, a sequential Java Ray Tracing program [29] is ported to JAVM. The same setup in Section 6.2 is used.

The sequential version is run on a standalone machine to generate a scene of 512 x 512 pixels. Time ($\mathbf{T_{seq}}$) needed to run this program is **0.586x 10^6 milliseconds**.

The JAVM-ised version breaks up the scene to be generated into smaller sub-regions and distribute them to the volunteers so that the generation can be performed in parallel. Time ($\mathbf{T_{dis}}$) is measured from the moment the client creates the JAVM environment to the moment the client collects back all the computed results from the volunteers. The speedup ($\mathbf{T_{seq}/T_{dis}}$) is then computed. The experimental results collected are as shown in Table 6.3.

No. of Volunteers (n)	**Time Taken (T_{dis}) (x 10^6ms)**	**Speedup**
2	0.297	1.976
4	0.181	3.240
8	0.094	6.220
16	0.053	11.140
32	0.034	17.106
64	0.029	20.117

Table 6.3. Ray Tracing Experimental Results

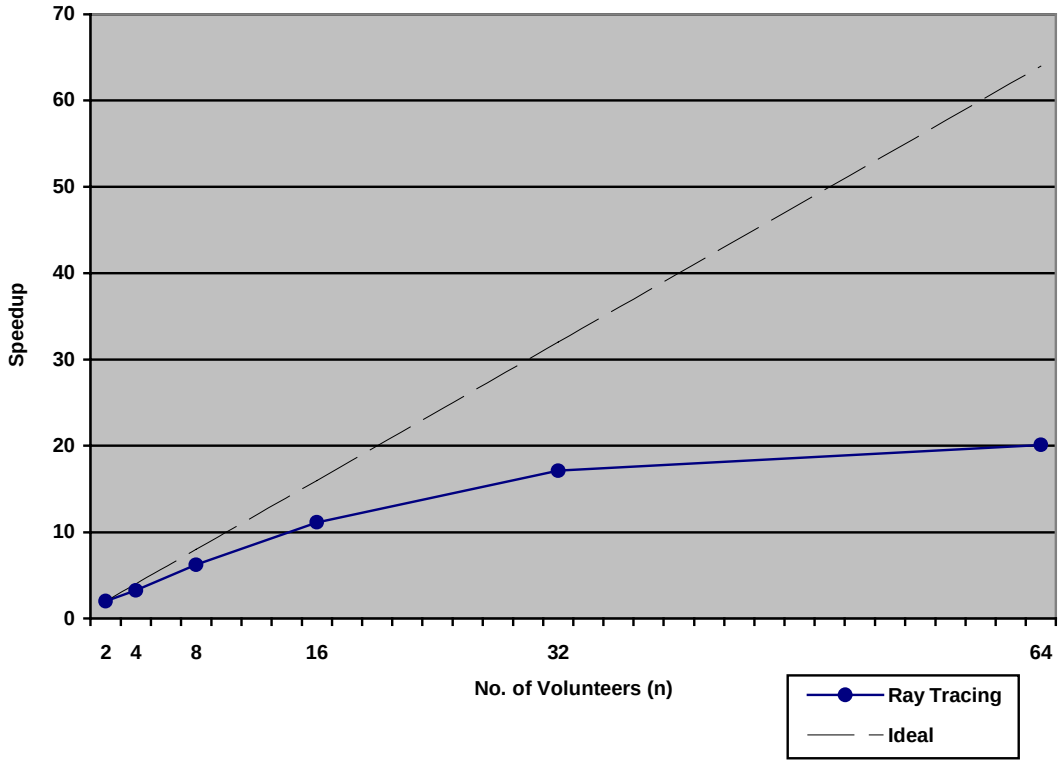

Figure 6.3. Ray Tracing Experimental Results

6.3.2 Performance

Unlike the previous case, from Figure 6.3, the speedup for JAVM-ised Ray Tracing starts to level off with 32 processors. The characteristics associated with this application are:

- Not as computationally intensive as RC5 cracking, basing on the time required for executing the program.
- Low bandwidth requirement. Data exchanges among entities are not too high.

6.4 Matrix Multiplication

The multiplication of two matrices is one of the most basic operations of linear algebra and scientific computing, and has provided an important focus in the search for methods to speed up scientific computation. The algorithm for performing matrix multiplication is fundamentally very simple and involves a series of multiplications and additions. A typical example of matrix multiplication is as follows:

$$\begin{pmatrix} A_{11} & A_{12} \dots A_{1M} \\ A_{21} & A_{22} \dots A_{2M} \\ \dots & \\ A_{N1} & A_{N2} \dots A_{NM} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \dots B_{1N} \\ B_{21} & B_{22} \dots B_{2N} \\ \dots & \\ B_{M1} & B_{M2} \dots B_{MN} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \dots C_{1N} \\ C_{21} & C_{22} \dots C_{2N} \\ \dots & \\ C_{N1} & C_{N2} \dots C_{NN} \end{pmatrix}$$

where $C_{11} = A_{11}*B_{11} + A_{12}*B_{21} + \dots + A_{1M}*B_{M1}$

$C_{12} = A_{11}*B_{12} + A_{12}*B_{22} + \dots + A_{1M}*B_{M2}$

$\dots$

$C_{NN} = A_{N1}*B_{1N} + A_{N2}*B_{2N} + \dots + A_{NM}*B_{MN}$

Figure 6.4. Matrix Multiplication

6.4.1 Experimental Results

The sequential version is run on a standalone machine to perform matrix multiplication on two square matrices of size 128 x 128. Time ($\mathbf{T_{seq}}$) needed to run the program is measured. This value is **141 ms**.

A number of parallel algorithms have been developed to perform matrix multiplication in an efficient manner [30]. However, in our experiments for performing parallel matrix multiplication on the JAVM system, efficiency is not the main objective. Instead, we hope to investigate the impact of transferring large amount of data between entities (in particular, client and volunteers) on the JAVM performance.

In the parallel program that we developed, matrix A is broken up into groups of equal-sized rows, basing on the number of volunteers that are used. During computations, each sub-matrix A of certain rows and the entire matrix B are transferred to a volunteer to generate the results for the same rows in matrix C. All computed results would be sent back to the client to form matrix C.

Time ($\mathbf{T_{dis}}$) is measured from the moment the client creates the JAVM environment to the moment the client collects back all the computed results from the volunteers. The speedup ($\mathbf{T_{seq}/T_{dis}}$) is then computed. The experimental results collected are as shown in Table 6.4.

No. of Volunteers (n)	Time Taken (T_{dis}) (x 10^3ms)	Speedup
2	3.11	0.045
4	3.593	0.039
8	5.688	0.025
16	8.000	0.018
32	13.219	0.011
64	22.781	0.006

Table 6.4. Matrix Multiplication Experimental Results

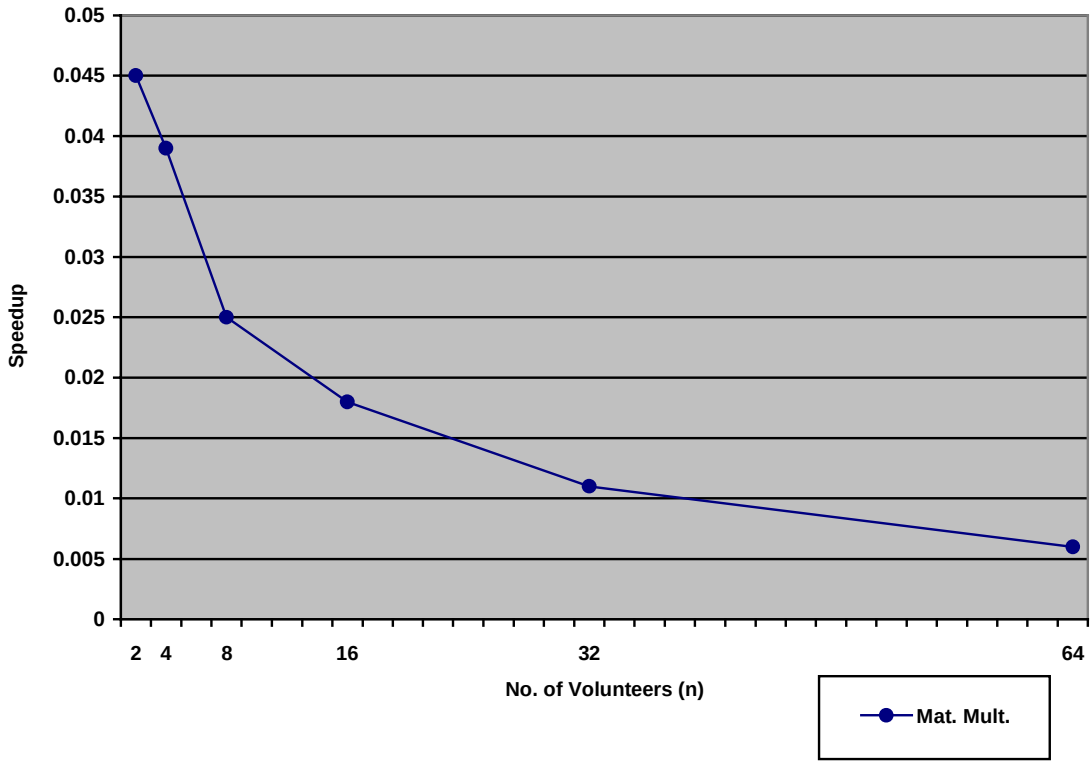

Figure 6.4. Matrix Multiplication Experimental Results

6.4.2 Performance

From Figure 6.4, the performance for JAVM-ised matrix multiplication is very poor. Not only there is no performance improvement, the speedup value decreases with the use of more volunteers. Characteristics of this application are:

- Low computation requirement. The size of the matrices used in experiments is not very large.
- High bandwidth requirement. The amount of data for a matrix of size 128 x 128 to be transferred from the client to a volunteer is already 1 Mbits. Hence, as more volunteers are involved in the computation, the network throughput at the client's end becomes a performance bottleneck.
- Long encryption and decryption time. In the JAVM system, the communication exchanges among the entities are encrypted. As the amount of data involved in the exchanges is huge, time taken for encryption and decryption of the data would be significantly high.

Experiments are conducted to measure the time for transferring data from client to a volunteer and for the volunteer to perform the necessary decryption, during task allocation. From our measurements, these overheads take up about 25% of the overall processing time.

6.5 Remarks

From the above experiments, the speedup of a JAVM-ised application depends largely on the computation to communication ratio involved. For good speedup results, computational performance gained for executing an application in parallel with a group of volunteers must be able to offset the loss incurred by communications overhead. In general, JAVM is well suited for executing coarse-grained applications (such as RC5 cracking) with high computation to communication ratio.

7. Conclusions

7.1 Summary of Work Achieved

Despite the dynamic and unpredictable nature of Internet as a computational resource, the potential of utilizing it for metacomputing is still great. Java, with all the features that it can provide as a programming language for network applications, is an attractive tool for developing systems that can harness this resource. Presently, there are a number of Java-based implementations for parallel computing. In general, these implementations can be categorized into Java applet based and Java standalone application based. However, most of these systems are not designed with Internet-based computing in mind. As such, their deployments in the Internet environment may not be feasible.

JAVM aims to address deficiencies of the existing implementations for Internet-based parallel computations. Its implementation goals are *ease of use*, *heterogeneity*, *portability*, *security*, *fault tolerance*, *load balancing*, *scalability* and *accountability*. As of our knowledge, it is the only Internet-based computing system that supports dynamic code execution without the use of applets, network load scheduling and encryption of network communications. Preliminary testing based on the execution of parallel applications developed for the JAVM system has also turned out to be promising.

7.2 Future Directions

The initial objectives set for this project has been achieved. However, there are still several areas whereby JAVM could be further improved and enhanced.

1. **Code Obfuscation.** Much of the information of Java source code remains in the bytecode. As such, decompilation of Java class files into the sources codes can be easily performed. This threat of reverse engineering is even more serious in the JAVM system whereby application bytecodes are dynamically transferred to machines that are not trustworthy. To counter this, code obfuscation has to be applied to the application class files before the transfer takes place. Code obfuscator, such as Cream [30], is able to scramble the symbolic information in the class files, so that they become less vulnerable to decompilation. However, the functionality of the program is still maintained. In JAVM, besides playing a role in security, code obfuscation can also help to reduce the size of application bytecodes during task placement. However, performance overhead caused by code obfuscation should not be overlooked.

2. **Detection of Intentional Faults.** As mentioned in Section 4.1.4, at this point in time, JAVM does not provide mechanisms for the detection of intentional faults, such as submission of erroneous results by volunteers to clients. Once such faults are detected, there should also be mechanisms to recover from the faults automatically. Techniques, such as *replication* and *spot-checking* [3], adopted by some systems, though not foolproof, are worthwhile further investigations.

3. **Resource Usage Check and Control by Volunteer.** Presently, in the JAVM system, once a volunteer registers with a coordinator, there is no way for the volunteer to find out how much of its computing resource is being tapped by clients. Besides the amount of lease time for volunteering that it can specify during registration, there is also no other way to control the amount of its resource that clients can use at one time.

4. **GUI-based Monitoring at Coordinator.** A graphical-based representation of the status of all the volunteers registered with the volunteer would help in the monitoring and management of the JAVM system.

References

[1] A. Baratloo, M. Karaul, Z. Kedem, P. Wyckoff, "Charlotte: Metacomputing on the Web", 9th International Conference on Parallel and Distributed Computing Systems (PDCS), 1996. http://www.cs.ucsb.edu/research/superweb/

[2] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, D. Wu, "Javelin: Internet-Based Parallel Computing using Java", http://www.cs.ucsb.edu/research/superweb/home.html

[3] Luis F. G. Sarmenta, S. Hirano, S. A. Ward, “Towards Bayanihan: Building an Extensible Framework for Volunteer Computing using Java, http://www.cag.lcs.mit.edu/bayanihan

[4] P. A. Gray, V. S. Sunderam, “The IceT Framework for Metacomputing”, http://www.mathcs.emory.edu/~gray/abstract7.html

[5] Data Security. RSA’s Secret-Key Challenge Solved by Distributed Team in Record Time. http://www.rsasecurity.com/news/pr/980226.html

[6] L. Peh, “ The Design and Development of a Distributed Scheduler Agent”, Honours Year Project Report, National University of Singapore, 1994.

[7] G. C. Fox, “Computing on the Web: New Approaches to Parallel Processing Petaop and Exaop Performance in the Year 2007”, http://www.npac.syr.edu

[8] G. C. Fox, W. Furmanski, “Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modelling”, http://www.npac.syr.edu/users/gcf/01/terri/SCCS_793

[9] E. R. Harold, "Java Network Programming", O'Reilly & Associates, 1997.

[10] M. O'Connell, "New benchmark results show Java ready for prime time of servers", http://www.sunworld.com/swol-07-javabenchmark.html?072098a

[11] A. J. Ferrari, “JPVM: Network Parallel Computing in Java”, Technical Report CS-97-29, Dept. of Comp. Sci, Unverisity of Virginia, Charlottesville, VA 22903, USA, http://www.cs.virginia.edu/~ajf2j/jpvm.html

[12] David A. Thurman, “jPVM: The Java to PVM Interface, December 1996, http://www.isye.gatech.edu/chmsr/jPVM/

[13] L. Vanhelsuwe, “ Create your own supercomputer with Java”, http://www.javaworld.com/jw-01-1997/jw-01-dampp.ibd.html

[14] N. Yalamanchilli, W. Cohen, “Communication Performance of Java based Parallel Virtual Machines”, ACM 1998 Workshop on Java for High Performance Network Computing, http://www.cs.ucsb.edu/conferences/java98/program.html

[15] Luis F. G. Sarmenta, “An Adaptive, Fault-tolerant Implementation of BSP for Java-based Volunteer Computing Systems, http://www.cag.lcs.mit.edu/bayanihan

[16] S. Brody, “Can HotSpot jumpstart your Java applications?”, http://www.sunworld.com/swol-05-1999/swol-05-hs.html?0524

[17] P. Rao, "Using Java: Is Java Secure?", http://www.usenix.org/publications/java/usingjava12.htm

[18] MPIJ 1.1, http://ccc.cs.byu.edu/OnlineDocs/docs/mpij/MPIJ.html

[19] A. L. Ananda, G. Tan, L. F. Lau, "Distributed Scheduling Algorithms for the Astra Virtual Machine", ACSC'97 (Australian Computer Science Communications Volume 19, Number 1), 1997

[20] Java Cryptography and Security: JCSI Public Release 3, http://security.dstc.edu.au/projects/java/release3.html

[21] T. Sundsted, "In Java we trust", Java World January 1999, http://www.javaworld.com/javaworld/jw-01-1999/jw-01-howto_p.htm

[22] "JCA/JCE and the DSTC Security Provider", http://security.dstc.edu.au/projects/java/misc/dstc_provider

[23] J. Knudesen, "Java Cryptography", O'Reilly & Associates, 1998

[24] B. Milewski, "The battle of languages – Java vs C++", http://www.relisoft.com/java/c_java.html

[25] S.C. Chan, "An Overview of the Java Security", http://home.hkstar.com/~alanchan/papers/javaSecurity/index.html

[26] Sun Microsystems, "The 100% Pure Java Initiative White Paper", http://java.sun.com/100percent/wp.html

[27] R.L. Rivest, "The RC5 encryption algorithm", CryptoBytes, 1(1): 9-11, 1995

[28] G. Hewgill, "RC5 and Java Toys", http://www.hewgill.com/rc5/index.html

[29] M. Armstrong, Y. Ma, "Java Ray Tracer", http://robotics.eecs.berkeley.edu/~mayi/CS184/

[30] S. C. Chan, "An Overview of the Java Security", http://home.hkstar.com/~alanchan/papers/javaSecurity/index.htm

[31] Search for Extraterrestrial Intelligence (SETI) Project, http://seti@home.ssl.berkeley.edu